The National Virtual Observatory Book
ASP Conference Series, Vol. 382, © 2008
M. J. Graham, M. J. Fitzpatrick, and T. A. McGlynn, eds.

Chapter 59: Web Services

Matthew J. Graham

Introduction

Web services are a cornerstone of the distributed computing infrastructure that the VO is built upon yet to the newcomer, they can appear to be a black art. This perception is not helped by the miasma of technobabble that pervades the subject and the seemingly impenetrable high priesthood of actual users. In truth, however, there is nothing conceptually difficult about web services (unsurprisingly any complexities will lie in the implementation details) nor indeed anything particularly new.

A *web service* is a piece of software available over a network with a formal description of how it is called and what it returns that a computer can understand. Note that entities such as web servers, ftp servers and database servers do not generally qualify as they lack the standardized description of their inputs and outputs. There are prior technologies, such as RMI, CORBA, and DCOM, that have employed a similar approach but the success of web services lies predominantly in its use of standardized XML to provide a language-neutral way for representing data. In fact, the standardization goes further as web services are traditionally (or as traditionally as five years will allow) tied to a specific set of technologies (WSDL and SOAP conveyed using HTTP with an XML serialization). Alternative implementations are becoming increasingly common and we will cover some of these here. One important thing to remember in all of this, though, is that web services are meant for use by computers and not humans (unlike web pages) and this is why so much of it seems incomprehensible gobbledegook.

In this chapter, we will start with an overview of the web services current in the VO and present a short guide on how to use and deploy a web service. We will then review the different approaches to web services, particularly REST and SOAP, and alternatives to XML as a data format. We will consider how web services can be formally described and discuss how advanced features such as security, state and asynchrony can be provided. Note that much of this material is not yet used in the VO but features heavily in IVOA discussions on advanced services and capabilities.

1. Web Services in the VO

Much can be achieved with simple services that consist of an HTTP GET call to a CGI script or Java servlet, e.g. Cone Search and SIAP. There are, however, several instances in the VO where these are not sufficient: for example, when the input data to a service is more complicated and structured than just a few simple key-value pairs, e.g. an array or hierarchical data, or programmatic representations of the input and output data (code bindings) are desired. In such cases, a web service is warranted.

The most frequently used web services in the VO are the SkyNodes (see Chapter 54) which allow astronomical databases to be queried with the XML form of ADQL (see Chapter 36). The Open SkyQuery portal (see Chapter 13) manages distributed queries across multiple SkyNodes to support such operations as crossmatching. Whilst it is normally accessed through its web pages, it also has a web service interface so that its functionality can be called from a programmatic client. This is quite a common design for web-based tools with web pages providing a human interface and web services one for computers. Open SkyQuery also illustrates the concept of *web service composability* under which web services (in this case SkyNodes) are aggregated and the aggregation is then exposed as a web service in its own right. In fact, the WESIX application (see Chapter 14) aggregates Open SkyQuery with Sextractor to provide a web service that can identify sources on an astronomical image and then crossmatch them against known catalogs.

Both the Footprint Services (see Chapter 9) and STOMP (see Chapter 10) tools provide web services (and web pages) to determine the coverage and geometry of astronomical data sets and Boolean operations, e.g. intersection and union, on multiples of these. Footprint Services also supports STC (see Chapter 37) representations of such spatial regions which makes it a good service to aggregate with. For example, one could provide a service that crossmatches data sets with complex spatial geometries and identifies objects not detected due to lack of coverage.

VO web services are not just limited to catalog-related activities. The Spectrum Services (see Chapter 17) tool allows searching and manipulation of spectral data. One could derive a composite spectrum from all the SDSS elliptical galaxy spectra and then fit its continuum using a theoretical template. Actually the availability of the entire IRAF system as a web service (see Chapter 32) makes possible virtually any optical or infrared data analysis task, both on its own and also as part of a composite service via aggregation.

A number of VO infrastructure components are also accessible via web services. The new Registry Interface specification (see Chapter 43) defines two web service interfaces for compliant registries: one for searching and one for harvesting. The VOSpace specification (see Chapter 40) for a data storage interface deals purely with web services.

Other VO web services not covered elsewhere in this book include the CDS web services (`http://cdsweb.u-strasbg.fr/cdsws.gml`) giving access to the CDS holdings and support services and VOServices (`http://VOServices.net`) which provides a front end to certain NED functionality and services to calculate distance measures in a given cosmology.

2. How To Use and Build a Web Service

Web services can be built and used in virtually any language that supports network communication and XML processing. Most common languages, however, make life easy us for us by providing libraries and frameworks (see Table 1) to handle the details of web service technologies, leaving us free to concentrate on the code that actually makes use of a web service or that we're exposing as a web service. In this section we are going to show how to use and build web services in Java using the

Apache Axis library (which is included on the companion CD). As a first step, we need to setup our environment in the usual manner (see the Software section of the Introduction to this book):

```
> source $NVOSS_HOME/bin/setup.csh
```

2.1. Using a Web Service

Using a web service in a piece of code (known technically as *consuming*) should be no more difficult than calling a library routine and as transparent. Most web service frameworks have a tool which will parse the formal description of the web service (called the WSDL and described in Section 6) and generate a proxy for the service from it called a *stub*. Operations can be performed on the stub as though it were the actual service, e.g. calling particular methods, and the stub code takes care of converting these calls into a suitable format that can be sent to the remote service, the actual sending, the receiving of responses and the translation of these back into something that looks familiar and can be handled by the invoking code.

Table 1. Web service frameworks

Language	Package or library
Java	Apache Axis (http://ws.apache.org/axis)
	XFire (http://xfire.codehaus.org)
C#	.NET (http://msdn2.microsoft.com/en-us/netframework/default.aspx)
	Mono (http://www.mono-project.com)
Perl	SOAP::Lite (http://www.soaplite.com)
Python	SOAPPy/ZSI (http://pywebsvcs.sourceforge.net)
C++/C	gSOAP (http://www.cs.fsu.edu/~engelen/soap.html)
Ruby	soap4r (http://dev.ctor.org/soap4r)

As an example, we will call one of the VOServices web services to calculate the comoving line-of-sight distance for a given redshift and cosmology. The web service itself is implemented in C# using the .NET framework but that does not matter to us so we will call it using Java (although this could just as easily be Perl, Python, or any other language). This language independence between services and their clients is one of the advantages of web services.

First we need to generate the stub code for the web service from its WSDL file. To do this, we will use a tool in the Apache Axis library:

```
> java org.apache.axis.wsdl.WSDL2Java
        http://voservices.net/Cosmology/ws_v1_0/Distance.asmx?wsdl
```

where the URL is for the WSDL file. This produces a number of classes in a subdirectory (`edu/jhu/pha/skyservice`) under the current working directory. In particular, for each service a Java interface (`Distance.java`) and an implementation of the interface (called the *service locator*) (`DistanceLocator.java`) are generated. We will need to remember to import all these classes into our code:

```
import edu.jhu.pha.skyservice.*;
```

To consume the service in our code, we instantiate the locator and get a stub:

```
Distance service = new DistanceLocator();
DistanceSoap stub = service.getDistanceSoap();
```

and then we can call its methods:

```
float answer = stub.comovingLineOfSight(1.3, 0.7, 0.3, 0.7);
```

Other examples of this approach can be found in the chapters on Open SkyQuery, WESIX and STOMP.

In the above example, we generated a static representation of the web service on the client which we then used as a proxy for the service. There are other ways of calling web services that do not require this and these are discussed in more detail in Section 6.3.

2.2. Building a Web Service

If we want to write an implementation of a web service whose WSDL file already exists, e.g. it is an IVOA standard, we can follow a similar approach to consuming a service. The same packages and libraries that can produce a stub will often also produce the server-side code (called a *skeleton*) to handle the web service request and response. As an example, let's say that we want to write our own code to calculate comoving line-of-sight distances but we want to use the VOServices WSDL file so that any client we've already written to use that web service can be used with ours. Again we use a tool in the Apache Axis library to generate the skeleton code from the WSDL:

```
> java org.apache.axis.wsdl.WSDL2Java ––server-side  --skeletonDeploy
      true http://voservices.net/Cosmology/ws_v1_0/Distance.asmx?wsdl
```

As before, this produces a number of classes in a subdirectory (`edu/jhu/pha/skyservice`) under the current working directory. In particular, for each service, a skeleton class (`DistanceSoapSkeleton.java`) and an implementation template (`DistanceSoapImpl.java`) are generated. As it stands, the template does not do anything but we can code in the details of what each method should do, i.e. how the various cosmological distances should be calculated, and, after compiling it all, we will be ready to deploy our web service.

Before describing web service deployment, however, we will take a look at the other web service scenario in which we already have code that does something but there is no existing WSDL file for a similar service that we can utilize. As an example, let's imagine that we have a Java class (`MyServiceTempImpl.java`) which has a method (`sqrt`) that returns the square root of a supplied number:

```
public float sqrt(float arg) {
   return Math.sqrt(arg);
}
```

If we have a Java interface (e.g. `MyService.java`) that this class implements then we can generate a WSDL file from this (making sure that we have compiled the interface first of all) for our code using a tool in the Apache Axis library:

```
> java org.apache.axis.wsdl.Java2WSDL –o myservice.wsdl
   -l"http://localhost:8080/axis/services/MyService"
   -n "urn:MyService" –p"edu.somewhere.mycode" "urn:MyService"
   edu.somewhere.mycode.MyService
```

where `–o` indicates the name of the output WSDL file, `–l` indicates the location of the service (where it will be deployed), `–n` is the target namespace of the WSDL file, `–p` describes the mapping from the Java package name of our code to the target namespace and the final argument is our Java interface. Now we need to generate the skeleton for our code from the WSDL:

```
> java org.apache.axis.wsdl.WSDL2Java –o . –d Session –s –S true
   -Nurn:MyService edu.somewhere.mycode MyService.wsdl
```

where `–o` specifies where the skeleton classes will be generated, `–s` and `–S` are just shorthand for `--server-side` and `--skeletonDeploy` respectively, `-N` defines the mapping from the namespace to the package name and `–d` indicates how the service will be deployed ("Session" means that the server will create a new object for each session-enabled client who accesses the service). As in the previous scenario, an implementation template is generated which we will need to modify by adding the code we have already written (i.e. the contents of `MyServiceTempImpl.java`) and then compile all the code.

Whichever build scenario we've followed, we are now ready to deploy our web services. We're going to expose them to the world using the Axis server which is already installed in the Tomcat server that we include on the companion CD so we need to ensure that Tomcat is running: `starttomcat`. One of the files created when the skeleton for the web service was generated is the *Web Service Deployment Descriptor* (WSDD) file which is an XML file containing information about the service and how it plugs into the Axis framework (note that WSDD files are peculiar to Apache Axis and not a standard web service feature). Deploying the web service is just a matter of calling another tool in Axis:

```
> java org.apache.axis.client.AdminClient deploy.wsdd
```

where `deploy.wsdd` is the WSDD file for our service. We should now be able to call our web service at: `http://localhost:8080/axis/services/MyService`.

3. Web Service Models

Having dealt with the practical side of consuming and deploying a web service, the rest of this chapter is devoted to the more technical and technological aspects of web services. As a prelude to the review of the different approaches to web services (see Sections 4, 5 and 8.6), it is worth saying a little about their general architecture. Conceptually, a web service can be decomposed into a combination of identifiable entities, actions, messages and constraints. Associated with these are the Resource, Service, Message, and Policy models for the web service respectively.

Different approaches to web services (and their accompanying technologies) focus on a specific model and bring its features to the fore: the aspects covered by the other models are present in a service implementation but remain largely in the background. For example, a service-oriented approach concentrates on the actions of a web service. Messages, identifiable entities and constraints are still present but the emphasis is on what the service does and not what it is acting upon, any sequence of messages which caused the action or any particular limitations of the action. Consequently, WSDL (see Section 6), which is a service-oriented technology, focuses on describing the exposed operations of a web service and requires extensions to handle things like message sequencing, resource management and security policy (see Section 8).

By identifying the predominant features of a web service in this way, the best approach to implementing it can be found. A service which deals with database records requiring just create, delete, update and retrieve functionalities may be better served with a REST interface than one based on WSDL and SOAP.

3.1. Resource-oriented Model

Generally, a resource is anything that can have an identifier but in the context of web services, it can also be owned and thus have policies set on it, e.g. management or access security. A web service itself is a resource. This model also focuses on resource descriptions – machine-readable data used to discover the resource (for example, a resource record in a VO registry) – and resource representations – data reflecting the state of a resource. For example, a resource that is an observation of a galaxy might have different image representations: FITS, jpeg, gif, each of which reflects the state of the resource at the time that representation was generated. Two subsequent representations might differ if some data processing has occurred on the resource in between them being generated thus reflecting a change in the state of the resource.

3.2. Service-oriented Model

A service is something capable of performing a piece of coherent functionality. It is realized by a provider agent – a piece of software such as Apache Tomcat - and used by a requester agent such as a web service client. This model focuses on the actions that may be performed by an agent, i.e. the nuts and bolts of a service: how they are described, the choreography of messages exchanged between agents (such as requester and provider), and the successful completion of an action.

3.3. Message-oriented Model

A message is the basic unit of data exchanged between agents and this model focuses on message structure and transport. It is not concerned, however, with any semantic aspect of a message: the reasons for sending it, the meaning of its contents, any actions taken in response to receiving it or the choice of transport protocol.

3.4. Policy-oriented Model

A policy is a set of assertions expressing capabilities and constraints. There are two fundamental types: obligations (actions that an agent is required to do) and permissions (actions that an agent can perform successfully). This model focuses on aspects such as security, quality of service and management which are most easily expressed as policies.

4. REST

One simple approach to web services is to use the basic infrastructure of the web in a very straightforward fashion. This immediately allows us to take advantage of the enormous proven scalability of the web architecture. The web follows a resource-oriented model (see Section 3.1): resources (web pages, media files, etc.) are identified by URIs, resource representations are communicated in many data formats (e.g. HTML, JPEG, PNG) and there are protocols (e.g. HTTP, FTP) that support interaction between agents (e.g. web browsers) and resources. Roy Fielding constrained this into an architectural style for web services known as REST (*Re*presentation *S*tate *T*ransfer: Fielding 2000): each resource has a URI, resource representations are exchanged in XML over HTTP and agents (services) employ the HTTP methods (GET/POST/PUT/DELETE) as a standard API to ensure uniform interface semantics (see Table 2).

An analogous mechanism is `/proc` under the Unix file system. Every process (resource) running on a Unix machine has a corresponding subdirectory in /proc identified by its process id (pid) and containing information (representation) about the process. This information can be retrieved (HTTP GET) with standard tools such as '`cat`' so the current status of a process with `pid=123` could be examined with: `cat /proc/123/status`. When a new process is started, an entry is created in `/proc` with its pid and its new state information but this happens as a consequence of executing a command. Under REST, a new job would be initiated by doing an HTTP PUT of a resource representation to a web service.

REST distinguishes resources and operations as separate concerns with the former associated with the HTTP URL and the latter with the HTTP method. In the real world, however, things are rarely so clean and simple and it is common practice with HTTP GET to conflate identification and functionality by using parameters (or verbs) in the URL, e.g. `http://processes.com/services?action=getStatus&pid=123` instead of the RESTian `http://processes.com/123/status`. The problem with this practice is that (under REST at least) HTTP GET requests are supposed to be *idempotent* which means that the effect of one or more identical requests

should be the same. By allowing parameters into the URL, it is possible to create situations where this is no longer true: for example, `http://processes.com/services?action=cancelJob&pid=123` will cancel the job the first time it is called but subsequent calls will have a different effect (and maybe even throw an exception) as there is no longer any job with this pid to cancel. This effectively negates the uniform interface semantics that lie at the heart of REST and, in fact, is really a service-oriented approach since what is now important is the action of the HTTP GET request defined by its parameters.

Web services that maintain idempotency whilst allowing parameterized URLS are known as "accidentally RESTful". Such services actually constitute the vast majority of web services which claim to be RESTful. The danger with these is that one assumes a full RESTful architecture based on the semblance of RESTful behavior and tries to do an HTTP POST to update a resource or an HTTP DELETE to cancel a job and gets an exception back or worse. Nevertheless accidentally RESTful services are very successful. Very little new infrastructure is actually needed in comparison to other web service styles to implement one, just HTTP and XML processing technologies, which are pretty much standard kit these days. They can be called from the simplest of clients, e.g. wget or xsltproc and they are commercially popular (see Table 3 for some well-known service implementations). If the figures are to be believed, 85% of web service traffic goes to such services and they are typically six times faster than other style implementations.

Table 2. The REST API.

HTTP Protocol	Action	Description
PUT	CREATE	Create a new resource
GET	RETRIEVE	Retrieve a resource representation
POST	UPDATE	Update a resource
DELETE	DELETE	Delete a resource

Table 3. Popular "accidentally RESTful" web services.

Organization	URL for service details
Amazon	http://www.amazon.com/gp/aws/landing.html
Yahoo	http://developer/yahoo.net
eBay	http://developer.ebay.com/rest
Flickr	http://www.flickr.com/services
YouTube	http://www.youtube.com/api2_rest
Del.icio.us	http://del.icio.us/doc/api

Another contributing factor to the current interest in REST-style services is the *AJAX* phenomenon. AJAX (Asynchronous JavaScript and XML) is a set of technologies that is used to make web pages and applications more responsive by putting a middleware layer (an AJAX engine written in JavaScript) into the web browser. This allows data exchange (normally in XML but the usage of other formats, e.g. JSON, is

increasing and see Section 7) with the server independent of user activity. So data can be preloaded and processed behind the scenes. Google Maps is perhaps the best known AJAX application. Using AJAX to call REST-style services transforms the browser into a web service client without the need for new technology or infrastructure since it is utilizing the browser's built-in support for XML processing (*DOM* and *XSLT*).

REST is purely an architectural style and RESTful services can easily be implemented with existing web application frameworks, i.e. CGI, Java servlets, etc. The most recent versions of some of the web service frameworks listed in Table 1 now support RESTful services, though this tends to be just ensuring that all four HTTP methods are available and providing simple tools for resource management. There are also a growing number of toolkits to support the development of RESTful services, particularly resource management, and two of particular note in Java are: Restlet (`http://www.restlet.org`) and REST-art (`http://rest-art.sourceforge.net`).

4.1. When to Use a RESTful Service

If the majority of web services are some flavor of RESTful and they are easy to use and implement then why bother with other styles? The pragmatic answer is that REST-style services should be used when the type of functionality required is akin to what the WWW offers, i.e. point a browser at a resource and get it. REST-style services have no formal method for describing the web service and so dynamically configuring a generic proxy or generating static client stub code for them is very difficult: for a pure REST service, however, the interface is just the standard one (HTTP methods) so this is not a problem. REST has no messaging infrastructure to support reliable messaging (ensuring that a message has been received) or message routing (specifying the route a message will take or addressing it), although being a resource-oriented approach, the emphasis is more on whether a resource representation has been successfully transferred rather than whether a message has got through. There is also no support for message-level security such as digital signatures and it is assumed that transport-level security (HTTPS) is sufficient. REST-style services also have no support for resource lifecycle management, transactions, attachments or asynchronous event notification.

5. SOAP

SOAP (Simple Object/Service-Oriented Access Protocol) is an XML-based messaging framework intended for exchanging information between peers in a decentralized, distributed environment and forms the basis of the W3C's Web Services Architecture Stack. It defines the message structure but not the message content and so needs to be combined with other technologies for a full web service implementation. SOAP-based web services are the main alternative to REST-style ones and, depending on what SOAP is used with, are either service-oriented (the vast majority at present) or message-oriented (a small growing fraction). To illustrate the difference between the two, consider a web service with a method which takes a VOTable data set and saves it. In the service-oriented case, the method will receive an XML document that it will need to parse and verify as being a VOTable before writing it out to disk. In the mes-

sage-oriented case, the method will receive a SOAP message containing a VOTable. The message headers will need to be processed before the VOTable can be extracted and saved and a response message sent back to the originator of the message request. In both cases, SOAP is employed for the messaging but with different web service technologies to provide the rest of the functionality.

A SOAP message consists of a SOAP envelope that encloses two data structures - the SOAP header and the SOAP body – and specifies the XML namespace and schema to be used for it. The SOAP header is optional but when present it contains information about the contents of the SOAP body, such as routing, transactional, security, contextual or user profile information. The SOAP body contains the actual data (message payload) to be consumed and processed by the receiver of the message. The data is encoded in an XML tag-based representation and the SOAP framework provides rules which describe how application-defined data types are to be mapped into XML tags, for example, a Java int may be represented by an XML element of type xs:integer.

SOAP is fundamentally stateless (i.e. it has no memory of what has happened previously) and a one-way message exchange paradigm but SOAP messages can be combined to create exchange patterns such as request/response. An example is given here:

```
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope">
  <soap:Body>
    <ComovingLineOfSight xmlns="http://www.skyservice.pha.jhu.edu">
      <z>float</z>
      <hubble>float</hubble>
      <omega>float</omega>
      <lambda>float</lambda>
    </ComovingLineOfSight>
  </soap:Body>
</soap:Envelope>
```

This above is the SOAP request message to a web service which will calculate the comoving line-of-sight distance for a given redshift and cosmology. The response containng the answer is:

```
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope">
   <soap:Body>
      <ComovingLineOfSightResponse
                        xmlns="http://www.skyservice.pha.jhu.edu">
         <ComovingLineOfSightResult>float</ComovingLineOfSightResult>
      </ComovingLineOfSightResponse>
   <soap:Body>
</soapEnvelope>
```

This exchange is semantically equivalent to calling a method with a method signature of `float ComovingLineOfSight(float redshift, float hubble, float omega, float lambda)`.

6. WSDL

WSDL (Web Services Description Language) is an XML grammar for describing the public interface of a web service in terms of its exposed operations and the message formats and protocol bindings required to interact with them. It is most commonly used in conjunction with SOAP and defines the format of the message content in the SOAP body. The WSDL file for a service contains everything that is needed to use the service.

6.1. Anatomy of a WSDL File

Before dissecting a WSDL file, it is worth identifying the key parts. Each WSDL describes one or more `services` which exchange `messages`. The data in these messages is defined in a set of `types`. Each message transfer in a service defines an `operation`. Operations have `bindings` to specific implementations using web protocols like HTTP.

In detail, a WSDL document has the following structure:

```
<definitions name="DefinitionsName"
   targetNamespace="NamespaceURI"
   xmlns:prefix="NamespaceURI"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns="http://schemas.xmlsoap.org/wsdl/">
```

This is the root element of the WSDL document and defines the various namespaces used in the WSDL, particularly the `targetNamespace` which is the namespace for the message contents of the SOAP body and the namespace `prefix` used to refer to it. For example, the VOServices cosmological distance web service called in Section 2.1 has `targetNamespace = "http://www.skyservice.pha.jhu.edu"` and `xmlns:s1` defined to be the same so the elements in the SOAP body will bear the namespace prefix `s1` mapped to this namespace.

```
   <import namespace="OtherNamespaceURI" location="URL"/>
```

This will import another WSDL document located at `URL` for use in this one. Its contents are defined to be in the `OtherNamespaceURI` namespace. This allows reuse of components designed for other web services and also makes web service composability easy: for example, a registry that supported both the search and harvest web service interfaces could describe this with a simple WSDL that imported the WSDLs for the two separate interfaces.

```
<types>
   <schema>…</schema>
</types>
```

This contains XML schemas (see Chapter 57) that define the datatypes that are used in the messages: for example, the elements holding the values of the cosmological parameters and the resultant distance. As with all XML schemas, datatypes (global elements and complexTypes) that will be used elsewhere are named.

```
<message name="MessageName">
   <part name="PartName" type="TypeNameReference"/>
</message>
```

This specifies the structure of the messages exchanged by the service (`ComovingLineOfSight` and `ComovingLineOfSightResponse` in the VOServices example). The `part` subelement identifies individual pieces of the message (e.g. `z`, `hubble`, and `lambda`) and their datatypes (*`TypeNameReference`* refers to the name given to the particular datatype in the `types` element).

```
<portType name="PortName">
   <operation name="OperationName">
      <input message="MessageNameReference"/>
      <output message="MessageNameReference"/>
      <fault message="MessageNameReference"/>
   </operation>
</portType>
```

This describes the set of operations (i.e. methods) that a web service *endpoint* supports. The endpoint is the URL at which the web service has been deployed and to and from which SOAP requests and responses are sent and received respectively. Each `operation` specifies its input message, output message and fault message, if any (*`MessageNameReference`* refers back to the name of one of the message elements). In the VOServices example, there will be an operation called `ComovingLineOfSight`.

```
<binding name="BindingName" type="PortNameReference">
   <soap:binding style="rpc|document"
      transport="http://schemas.xmlsoap.org/soap/http"/>
   <operation name="OperationName">
      <soap:operation soapAction="ActionValue"/>
      <input>
          <soap:body
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="TargetNamespace"
          use="encoded|literal"/>
      </input>
      <output>
          <soap:body
```

```
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
             namespace="TargetNamespace"
             use="encoded|literal"/>
         </output>
      </operation>
   </binding>
```

The specifications so far in the WSDL document have been abstract. The `binding` element describes what protocol (e.g. HTTP) will be used to send the SOAP messages over the wire (through the `transport` attribute of the `soap:binding` element). It also specifies how the input and output messages for each operation will be represented under that protocol, i.e. which rules will be used to translate programmatic datatypes into a XML tag-based representation. If the value of the `use` attribute in the `soap:body` element is `encoded` then rules identified by the URL value of the `encodingStyle` attribute will be applied. Normally this means that the messages can only use XML Schema datatypes. When the `use` value is `literal`, the datatype definitions specified in the user-defined `types` element of the WSDL be applied.

The `binding` element also defines how the structure of the SOAP message body should be interpreted. If the value of the `style` attribute of the `soap:binding` element is `rpc` then the endpoint should treat child elements in the body as a XML representation of a method call. The structure of the SOAP request body must contain both the method name and the set of its parameters. When the value is `document`, the structure of the body is unconstrained and can contain an arbitrary XML instance (modulo whatever datatypes are defined in the `types` element).

```
   <service name="ServiceName">
      <port name="PortName" binding="BindingNameReference">
         <soap:address location="URL"/>
      </port>
   </service>
```

The final element specifies the actual URL where the service is located. Each `port` element represents a single endpoint for the service tied to a particular `binding` (referred to by the *`BindingNameReference`*) so a web service could have multiple endpoints for load balancing or even have endpoints tied to a different protocol bindings.

```
</definitions>
```

WSDL styles (types) are referred to by the combination of the values of the `style` and `use` binding attributes: `rpc/encoded`, `rpc/literal`, and `doc/literal`. `doc/encoded` is never used but there is a fourth style known as `doc/literal wrapped` which is an undocumented pattern originating from Microsoft. The distinction between these styles can be illustrated by considering a hypothetical web service with an operation called *receive* which takes in an integer value. The `portType` for this operation in all styles is:

```
<portType name="foo">
   <operation name="receive">
```

```
      <input message="Request"/>
   </operation>
</portType>
```

Table 4 shows the different ways that the input message would be defined under the different WSDL styles and Table 5 shows the resulting SOAP message bodies for each type.

In `rpc/encoded`, the WSDL is as simple as it can be and the operation name appears in the SOAP message so the receiver knows what is being invoked; however, there is unnecessary type encoding information in the message and it is impossible to validate the SOAP message against any schema definition. The only difference between `rpc/encoded` and `rpc/literal` is that in the latter, the type encoding information does not appear in the SOAP message. In the `doc/literal` style, everything in the SOAP body is defined in a schema and so the message can be validated; the operation name, though, has been lost from the message so knowing what to invoke becomes difficult, if not impossible. Finally, in `doc/literal wrapped`, the SOAP message can be validated and the operation name appears but the WSDL is now quite complicated.

The `doc` style can pass an entire transaction as an XML document, is not constrained by `rpc`-oriented encoding, which carries a processing overhead with it in encoding payloads, and the message can be validated at call time. `doc`-style messages can be parsed using low memory XML parsers such as SAX and the style avoids `rpc`'s natural tendency to expose programming language object structures. This explains why 95% of web services use `doc/literal wrapped`.

There are, however, reasons not to use `doc/literal wrapped`. It does not support overloaded operations, i.e. when there is more than one method with the same name, such as *receive* taking an integer value and *receive* taking an integer and a float: the corresponding SOAP body element will be called `receive` in both cases and so it is unclear which operation is being referred to. It also has no standard way of representing data graphs (in fact, this is true of all `literal` styles) and so there is no guarantee that the client or service at the other end of the wire can interpret the graph structure properly. Finally it makes the WSDL much more complicated whilst producing exactly the same wire format as `rpc/literal`.

6.2. Writing WSDL

There are two approaches that one can take to generating the WSDL for a service: in *contract-last development*, the service is first implemented and then the WSDL is auto-generated from the service code using an appropriate software tool, e.g. Apache Axis for Java. Unfortunately this means that the contract (the messages that go across the wire between the service and a client) is tightly coupled to the service implementation's interface. If the implementation needs to be changed in any way then the WSDL file will almost certainly have to be regenerated, breaking any existing client code. In *contract-first development*, the semantics of the contract are designed and represented in WSDL and XML Schema (if necessary) and then server-side code stubs are generated from the WSDL and finally the business logic of the service is filled in. Since the details of the message exchanges have defined first, it is less likely

that the WSDL will have to be changed. Generally this is a more robust design style. The downside, of course, is that the WSDL document has to be written by hand but there are a number of good tools, such as oXygen and XMLSpy, which can make this process less grueling.

Table 4. Distinction in WSDL between different WSDL types.

WSDL type	WSDL

`rpc/encoded`	`<message name="Request">`
	`  <part name="x" type="xs:int"/>`
	`</message>`

`rpc/literal`	Same as `rpc/encoded`

`doc/literal`	`<types>`
	`  <schema>`
	`  <element name="xElement" type="xs:int"/>`
	`  </schema>`
	`</types>`
	`<message name="Request">`
	`  <part name="x" element="xElement"/>`
	`</message>`

`doc/literal wrapped`	`<types>`
	`  <schema>`
	`  <element name="receive">`
	`  <complexType>`
	`  <sequence>`
	`  <element name="x" type="xs:int"/>`
	`  </sequence>`
	`  </complexType>`
	`  </element>`
	`  </schema>`
	`</types>`
	`<message name="Request">`
	`  <part name="parameters" element="receive"/>`
	`</message>`

6.3. Using WSDL

The WSDL document contains everything that you need to know about a service to call it. From the perspective of what needs to be implemented in the underlying environment, the simplest way to invoke the service (called the *static invocation* or *binding model*) is, as we did in Section 2.1, to generate client-side code stubs from the WSDL using some appropriate tool, such as Apache Axis for Java, and then fill in the details of constructing and handling the request and response parameters in the programming language of your choice – the actual business of constructing, sending and receiving the SOAP message is taken care of by the generated stub code. Of course, if the WSDL changes then the client stubs will need to be regenerated and dependent code might also need to be rewritten.

Table 5. Distinction in SOAP messages between WSDL types.

WSDL type	WSDL

rpc/encoded	`<soap:body>`
	`  <receive>`
	`  <x xsi:type="xs:int">5</x>`
	`  </receive>`
	`</soap:body>`

rpc/literal	`<soap:body>`
	`  <receive>`
	`  <x>5</x>`
	`  </receive>`
	`</soap:body>`

doc/literal	`<soap:body>`
	`  <xElement>5</xElement>`
	`</soap:body>`

doc/literal wrapped	`<soap:body>`
	`  <receive>`
	`  <x>5</x>`
	`  </receive>`
	`</soap:body>`

An alternative way (called the *dynamic invocation* or *late binding model*) has no generated code but instead uses a proxy that dynamically builds a class instance at runtime that conforms to a particular interface described by the WSDL, proxying all service invocations to a single 'generic' method. This approach is much more flexible and greatly reduces maintenance costs as there is no client-side stub code to look after and a single client can be used with multiple different services. Apache Axis provides support for this mode of invocation. There are also a number of websites that make use of this technique, such as the Generic SOAP Client at `http://soapclient.com/soaptest.html`, to produce form interfaces to web services.

Finally an approach that can overcome the shortcomings of web service frameworks when they occur is to deal with the SOAP messages directly. Tools such as `wsdl` in the Mono toolkit can generate sample SOAP request and responses for a method from a WSDL, for example:

```
> wsdl –sample:ComovingLineOfSight
                          "http://voservices.net/Distance.asmx?wsdl"
```

These can then be edited by hand and submitted with tools such as `curl`:

```
> curl –s –H "Content-Type: text/xml; charset=utf-8"
   –H "SOAPAction: http://skyservice.pha.jhu.edu/ComovingLineOfSight"
   -d @request.xml "http://voservices.net/Distance.asmx"
```

where the `–H` arguments set HTTP header values in the request message and `request.xml` is the edited SOAP request message. The `Content-Type` and `SOAPAction`

values are part of the output of the `wsdl` command. The web service will reply with the appropriate SOAP response – in this case, containing the cosmological distance calculated from our input parameters.

6.4. Web Service Interoperability

Web services are supposed to be capable of interoperating, i.e. the same client can call different instances of a particular service or one service can call another service, without any difficulty. Unfortunately, it turns out that this not the case, which is why the Web Services Interoperability Organization (WS-I) was formed to provide a set of guidelines (called profiles) on how to be interoperable. WS-I Basic Profile defines the basic requirements and essentially says that `rpc/encoded` is not a compliant style. Note that the Java XFire library does not support `rpc/encoded` style web services for this reason. Tools to test whether a service conforms to WS-I profiles can be found on the WS-I web site (`http://www.ws-i.org`).

7. Data Formats

XML is text-based, platform-independent and capable of representing general data structures. This is why it is the primary technology employed in data exchange in web services. However, it can be syntactically verbose and incur significant processing overheads and so might not always be ideal. Fortunately there are a number of alternatives which are described below. Of these, only JSON currently has some use in the VO within NESSSI (see Chapter 20).

7.1. JSON

JSON (JavaScript Object Notation) is an increasingly popular lightweight data format (Yahoo added support for JSON in December 2005). It is a subset of the object literal notation in JavaScript but does not require JavaScript to be used. JSON represents data through the following basic types: number, string, Boolean, array, object, and null. It is also supported by many programming languages.

However, JSON has a number of limitations: there is no equivalent mechanism to XML Schema so validation of JSON files and data binding – generating object representations of the data in code - is not possible. The type system is also limited – for example, there is no representation of date or time structures – and does not allow extensions or versioning.

7.2. YAML

YAML (YAML Ain't Markup Language) is another up-and-coming lightweight data format, particularly as it is the de facto data serialization format in Ruby. All data are represented in YAML by combinations of lists, hashes (dictionaries) and scalars. In fact, YAML is essentially a superset of JSON (by accident rather than by design) and so YAML parsers can also be applied to JSON. There are bindings for YAML in a

number of languages and also at least one validator available for it (`http://www.kuwata-lab.com/kwalify`).

7.3. Microformats

Microformats are a set of simple data formats that can be embedded in *XHTML* web pages ensuring that machine and human readable data is maintained in a single document. Programs can easily extract the data from the web page and process it.

7.4. ATOM

In many cases, web services are working with time-stamped uniquely-identified data chunks with attached metadata. This is precisely the main use case for the ATOM Publishing Protocol. The Lucene Web Service (`http://dev.lucene-ws.net/wiki/API`) and Google Data (`http://code.google.com/apis/gdata/index.html`) APIs both offer ATOM interfaces.

8. Advanced Web Services

There are currently (late 2006) over 65 different specifications (collectively referred to as WS-*) describing extensions to the basic WSDL+SOAP model that address such features as workflows, security, messaging (reliable, publish-subscribe, choreography, etc), transactions, process and resource management, asynchrony and federation. Most of these work by defining additional elements in the SOAP message header and many have no reference implementation. WS-* is a two-edged sword that offers both increased functionality and complexity. Unfortunately, the latter tends to be far more successful at dissuading users than the former at attracting them. Faced by the WS-* plethora many people have abandoned SOAP-based services in favor of simpler alternatives such as REST. As already noted, however, these have their limitations and if you really require a particular piece of functionality, you might have to just bite the bullet and deal with one of the WS-* legion.

8.1. Attachments

It is clear that SOAP+WSDL can handle web services where the operation arguments are just basic datatypes but what happens when you also need to transfer a file to or from a web service? One solution is to stage the file somewhere easily accessible, e.g. an FTP site, and then pass the location as an argument in the web service call. The client or service is then responsible for transferring the data as a separate activity to the web service call. This works well for third-party transfers or where the file is large. Why, however, can't the file just be included with the SOAP message, in the same way that a file can be uploaded to a web site.

In fact, it turns out that there are two ways of doing this: *by value* and *by reference*. The by value method includes the file as part of the SOAP message body, e.g. as the value of a `<file>` element. If the file contains non-binary data then a straightforward XML representation of the file contents is possible. However, binary data

requires encoding so the file contents are represented as elements of either xs:hexBinary or xs:base64Binary types. Unfortunately this carries a data expansion by a factor of between 1.33 and 4, depending on the data type and encoding representation, and there are additional processing costs for encoding and decoding the data when sending and receiving the SOAP message containing it. Anything within the SOAP body gets parsed by the web services infrastructure so it might take a while to call a web service if you are including a 1 gigabyte data file in the SOAP message.

In contrast, the by reference method attaches the file as an external unparsed entity outside of the SOAP message and then uses a reference URI within the SOAP message to refer to it. There are three implementations of this approach: *SwA*, *DIME* and *MTOM*. SwA (SOAP with Attachments) uses the same technique as uploading a file from a web form and constructs a multipart MIME message from the SOAP message (part 0) and the file (parts 1 – n). `Content-Id` is used as the reference keyword within the SOAP part but there is a lack of length header on the message sections so it is impossible to know how big the attached file is. DIME (Direct Internet Message Encapsulation) follows a similar approach but uses a faster and more efficient binary encoding. Whilst DIME works in the field, Microsoft deprecated it in 2004 (in fact, it was never even a true standard) so it is hard to say how long it will continue to survive.

Unfortunately, both SwA and DIME introduce a data structure to the SOAP construct that is outside the realm of the XML data model: this means that there are no rules to specify how the attachment content relates to the SOAP envelope. This is incompatible with other WS-* specifications such as WS-Security that require this information to work. MTOM (Message Transmission Optimization Mechanism) gets around this by using XOP:Include as the reference mechanism (XOP is the W3C standard for XML Binary Optimized Packaging) so that conceptually binary data is base64-encoded within the SOAP XML document, even though it is actually present as a MIME construct (making it backward compatible with SwA). All the most recent web service frameworks for Java (Axis2, XFire) and C# (WSE 3.0) offer support for MTOM.

8.2. Addressing

Though it might be hard to believe, SOAP is, in fact, a very basic messaging framework: for example, it lacks a standard way to specify where a message is going, how to return a response or where to report an error. WS-Addressing is a W3C standard for incorporating message addressing information into the SOAP header, providing keywords such as: To, ReplyTo, FaultsTo, Anonymous, and MessageId / RelatesTo. It also provides a standard for including service-specific attributes. All the most recent web service frameworks for Java (Axis2, XFire) and C# (WSE 3.0) offer support for WS-Addressing.

8.3. Security

Web service security can be provided at two levels: *transport level* and *message level*. Transport-level security is the basis for secure web sites (using HTTP and SSL/TLS) and encrypts the entire communication between a sender and a receiver.

However, it only guarantees point-to-point security, i.e. from the sending port to the receiving port. If there are routers or other intermediaries between the client and the server, a sequence of secure links must be established between each sending/receiving pair. Each intermediary will decrypt any communication it receives and then reencrypt it as it sends it on to the next entity in the chain so only one link needs to be compromised for the entire system to be compromised. All connections also need to be kept persistent over the lifetime of an application session since the shutdown and recovery of an intermediary does not necessarily imply that the client-server communication can be recovered. Messages are only protected whilst in transit (i.e. on the wire) and so cannot be stored to later prove that they have not been tampered with. There is no support for *nonrepudiation* - an end-to-end audit trail from request to response which lets you prove that a user performed a certain action so that the user cannot deny it, e.g. the receiver of a message cannot legitimately claim that they didn't receive the message. Once a message has been received by a server, it is no longer secure (having been decrypted at the transport layer boundary) and so will be unprotected when passed on to other server layers (one hopes that the server itself has not been compromised). For example, if the server is just a system that invokes remote web services (such as a portal) then the client request will be unencrypted when it is passed onto the end service.

Message-level security only deals with the contents of the SOAP message and embeds all required security information in the SOAP message header. It guarantees end-to-end security, i.e. from the client application to the end web service, and is independent on any transport layer details and so is unaffected by intermediaries. Messages are protected for as long as the XML content is perceived as being a SOAP message and so can be stored and passed onto other layers with impunity. Message-level security also supports *data origin authentication*, i.e. the verification that the original source of a received message is as claimed. Transport-level security only handles *peer entity authentication* where the identity of a peer in an association, such as a connection between a sender and receiver, is the identity claimed. The level of security granularity offered by message-level security is also much finer since it can be applied to certain elements or fragments of the SOAP message, for example, different portions of a SOAP message can be encrypted and digitally signed by different individuals using different algorithms.

WS-Security is an OASIS (one of the organizations involved in defining the WS-*) standard for message-level security in web services. It supports the use of a number of different security tokens/credentials such as unsigned (username/password), binary (X.509 certificates) and XML (SAML token) to authenicate, sign and encrypt part or all of the SOAP message. Table 6 lists the web service frameworks offering support for WS-Security in a variety of languages.

It should be noted that working with WS-Security can be complicated. An alternative approach is to use WS-Policy and WS-SecurityPolicy which allow security requirements to be specified declaratively. There is support for WS-Policy in the latest framework for Java (Axis2, XFire) and C# (WSE 3.0) at least.

Table 6. WS-Security implementations

Language	Web service framework
Java	WSS4J (`http://ws.apache.org/wss4j`) used by AXIS2 and XFire
C#	WSE 3.0 (`http://msdn.microsoft.com/webservices/webservices/building/wse` WSRF.Net (`http://www.cs.virginia.edu/~gsw2c/wsrf.net.html`)
Perl	WSRF::Lite (`http://www.sve.man.ac.uk/Research/AtoZ/ILCT`)
Python	pyGridWare (`http://dsd.lbl.gov/gtg/projects/pyGridWare/`)
Ruby	wss4r (`http://rubyforge.net/projects/wss4r`)

.

8.4. State

Web services can either be *stateless* or *stateful*, depending on whether they retain any information about previous interactions with a particular client. Often stateless services are a better idea, They offer both reliability – in case of failure, the service can just be restarted without any concern about previous interactions – and scalability – new service instances can be arbitrarily created/destroyed in response to system load. However, there are circumstances when state is required such as dealing with interactive or asynchronous web services.

The best way to handle state is to separate the web service and the state information (known as a *resource* in this approach). A resource is identified by a unique key and message exchanges with the service are used to interact with the resource (manipulate state). There is a set of WS-* specifications – normally referred to by the main specification, WS-ResourceFramework or WSRF – which defines a framework for stateful services based on this approach.

Under WSRF, a stateful web service is a WS-Resource: an entity composed of a stateless web service and a stateful resource. The address of the WS-Resource is called an *endpoint reference* and is a quantity defined in the WS-Addressing specification. All operations on a WS-Resource are ACID-like: updates are made in all-or-nothing fashion (atomicity); there is a consistent state even after failure (consistency); updates are isolated within a given work unit (isolation); and updates are permanent (durability). WSRF also defines an extension to WSDL to specify the element that is representing the resource (state information).

Table 7 lists the web service frameworks that offer support for WSRF.

8.5. Asynchrony

The WWW mainly involves synchronous activities (although AJAX is going some way towards changing this): when something is submitted, a response is expected immediately or maybe with a few seconds' wait. Yet the scientific world (and, indeed, the real world) is far more asynchronous in nature, for example, retrieving and analyzing a large data set or running a simulation are activities that take time and do not happen instantaneously. In the context of web services, a distinction needs to be made between asynchronous messaging and actual asynchronous services.

Table 7. WSRF implementations

Language	Web service framework
Java	Globus Toolkit 4 (`http://www.globus.org`)
	Apache WSRF (`http://ws.apache.org/wsrf`)
C#	WSRF.Net (`http://www.cs.virginia.edu/~gsw2c/wsrf.net.html`)
Perl	WSRF::Lite (`http://www.sve.man.ac.uk/Research/AtoZ/ILCT`)
Python	pyGridWare (`http://dsd.lbl.gov/gtg/projects/pyGridWare/`)

The most frequently used *message exchange pattern (MEP)* is *request/response* where the endpoint receives a message and sends back a correlated message. For example, a client calls (sends a request to) a web service which calculates cosmological distances for a specified set of parameters and gets the answer back in response. Normally this is a fairly instantaneous activity but if the calculation is involved then the response could take a while to return and the client will just hang until it does so. However, if the client were to invoke the service asynchronously, for example, by sending the request over a one-way transport mechanism such as email or HTTP in one-way mode, then it could continue with other activities until it received the response from the service (again by email or HTTP). The fact that the request and the response are correlated in some fashion, e.g. by using a reference id in the messages, means that client can make correct sense of the message from the service when it gets it. Similar behavior can also be achieved with the other three MEPs that WSDL 1.1 defines – *one-way*, *solicit-response* and *notification*: for example, a one-way message could be sent from the client to the service with the parameters and a callback URL on the client and the service could respond with a notification message to the callback URL when the operation has completed.

An asynchronous service can allow more interaction than just taking input parameters and returning a result. It should be possible to at least check what the status of a particular activity is and cancel it, and maybe even pause and modify it. Unfortunately there are no current industry standards for asynchronous services but ASAP (Asynchronous Service Access Protocol) is being defined by OASIS and is now at committee draft status.

It is feasible to add asynchronous behavior to a web service without a standard and the web service frameworks that support state (see Table 7) are useful here. Java support for asynchronous invocation (messaging) can be found in Apache Axis2 and WSIF libraries (`http://ws.apache.org/axis2` and `http://www.apache.org/wsif`) and JMS (`http://java.sun.com/products/jms`).

The lack of any approved specification for a common interface to asynchronous services has led the IVOA Grid and Web Services Working Group to define their own pattern for asynchronous services known as the *Universal Worker Service (UWS)*. UWS proposes that a minimal interface supports job creation, polling of job status and retrieval of results. A fuller interface would also allow requesting an estimate of the duration of a job and restarting a failed job from its last checkpoint. UWS-PA is an instantiation of the UWS pattern for parameterized applications (PA), i.e. applications whose input and output arguments can be expressed as a set of key-value pairs. The Astrogrid Common Execution Architecture (CEA) is a reference implementation of UWS-PA.

8.6. Message Orientation

Although they are commonly paired with each other, SOAP and WSDL are actually largely orthogonal technologies – one is about messaging whilst the other is concerned with describing service interfaces. There is little true functional dependency between them. For example, WSDL can be used just as easily with HTTP POST and GET bindings as SOAP (in fact, this is the default behavior with .NET web services implementations).

There are a number of concerns that arise is using SOAP and WSDL together. WSDL is clearly an operation-centric technology and so focuses very strongly on interface abstractions to describe services (the so-called RPC mindset). It has limited modeling of interaction patterns in that the interactions it describes consist of no more than two messages within a single exchange. WSDL has no ability to capture choreographic information so it is impossible to specify ordering constraints between operations. It can also be difficult to describe infrastructure protocols that use SOAP headers so unfortunately most of the WS-* cannot be represented in a straightforward manner using WSDL. WSDL is also immutable: anything that is declared in it must be preserved until the associated web service is retired. Finally technologies that build atop WSDL have a tendency to be more verbose and complex than if other simpler contract languages were used.

MEST (*Mes*sage *T*ransfer) is an architectural style for web services that takes the Internet-centric philosophy of the REST approach and applies it in a message-oriented fashion. It has no notion of clients or servers just peers who can send and receive messages. Messages and services are first class abstractions (there are no concepts of interfaces, data or operations) and interactions between peers are described through MEPs that capture temporal and spatial information about the interaction with the arrival of a message at a service causing some processing to occur. MEST is largely time independent (asynchronous) as messages are delivered when a receiving peer is available and peer-to-peer relationships are not limited to one-to-one but can be one-to-many with the same message being duplicated and delivered to multiple peers. Constraints in timeliness and ordering are handled by the services themselves and are not considered part of the underlying architectural model which is deliberately simple.

SSDL (SOAP Services Description Language) is the MESTian equivalent to WSDL but is specifically focused on using SOAP as the messaging vector over arbitrary transport (and transfer) protocols. WS-Addressing is a first class citizen and used for embedding addressing information within SOAP headers and binding these addresses onto underlying transport protocols. SSDL also uses the same underlying component model as XML (XML Infoset) and XInclude for contract modularization. It is also very extensible in allowing arbitrary protocol frameworks to be defined specifying how the messages relate to each other.

A SSDL document has four components: *schemas* defining XML types used in the messages, *messages* defining the SOAP documents, *protocols* defining different protocol frameworks from MEPs (equivalent to what WSDL 2.0 offers) to sequential constraints, and *endpoints* which defines the WS-Addressing Endpoint Reference for the service. Unfortunately support for SSDL is still very limited with Soya (`http://soya.sourceforge.net`) as the only real implementation at present.

9. SOA and WOA

One of the most hyped concepts in the past few years has been *service-oriented architecture* (SOA) (note that this should not be confused with service-oriented web services discussed in 1.2). SOA is a design approach to distributed computing in which all functions are defined using a description language as independent services. These have invokable interfaces that can be called in defined sequences to form business processes. The principles behind SOA are:

Service reusability: as much as possible should be done with reuse in mind.
Service contract: services adhere to a contract that captures all the details of using the service and is formally described somewhere.
Service loose coupling: services have a minimally dependent relationship.
Service abstraction: no service logic is exposed to the world beyond what is described in the service contract.
Service composability: services can be aggregated to form composite services.
Service autonomy: services have control over the business logic that they encapsulate.
Service statelessness: services maintain minimal levels of state.
Service discoverability: services have descriptions that allow them to be found via discovery mechanisms.

The obvious components of a SOA infrastructure are SOAP, WSDL (or SSDL) and whatever web services frameworks are used to underpin the actual service implementations.

Web-Oriented Architecture (WOA) is a subset of SOA that advocates REST or POX over HTTP instead of SOAP for all the reasons that are usually given: ease of use, reliability, etc. In particular, a design style for robust scalable WOA clients (called *WOA/Client*) has been proposed, advocating that web services actually refers to anything over HTTP and should be completely agnostic about platform, technology, protocol, data format, contract description language and programming language. Clients should expect constant changes in the contract, protocol and endpoint and be deeply resilient with extreme fault tolerance and a mandatory pessimism about the quality of their runtime environment (the web). They should also be oriented to integrating data from many sources and re-serving it (mashup-oriented) but keep as much as possible to bare data formats thus avoiding transformation costs and lack of control.

References

Fielding, R. 2000, PhD Thesis, University of California Irvine,
http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Useful Links

IVOA Grid and Web Services Working Group. Nov 2006;
http://www.ivoa.net/twiki/bin/view/IVOA/IvoaGridAndWebServices#Asynchr

onous_activities_proposal [Accessed 30 Nov 2006]
SOAP v1.2 Primer. Jun 2003; http://www.w3.org/TR/soap12-part0/ [Accessed 30 Nov 2006]
WSDL Specification. Nov 2006; http://www.w3.org/2002/ws/desc/ [Accessed 30 Nov 2006]
WS-I. 2006; http://www.ws-i.org/ [Accessed 30 Nov 2006]
SSDL. Feb 2005; http://www.ssdl.org [Accessed 30 Nov 2006]
Apache Web Services Project. Nov 2006; http://ws.apache.org [Accessed 30 Nov 2006].
XFire. Oct 2006; http://xfire.codehaus.org [Accessed 30 Nov 2006]
Globus. http://www.globus.org [Accessed 30 Nov 2006]
WSRF::Lite. Oct 2006; http://www.sve.man.ac.uk/Research/AtoZ/ILCT [Accessed 30 Nov 2006]
pyGridWare. http://dsd.lbl.gov/gtg/projects/pyGridWare [Accessed 30 Nov 2006]
oXygen. Nov 2006; http://www.oxygenxml.com [Accessed 30 Nov 2006]
XMLSpy. Nov 2006; http://www.altova.com/products/xmlspy/xml_editor.html [Accessed 30 Nov 2006]